witnessCalc

A Tool for Calculating Responses of Witness Foils

By Robert Crabbs
9-22-2009

INTRODUCTION

In the interest of assaying spent nuclear fuel, we evaluate nuclear resonance florescence (NRF) as a possible means of identifying nuclides in a given sample. Every nuclide has a unique set of nuclear excitation energies, which can serve as a signature for the presence of that nuclide. In NRF interrogation, a continuous-spectrum photon beam is directed on a target. The photons that are very close to the excitation energies of the nuclides in the target are preferentially absorbed, and re-emitted a short time later as isotropic radiation of the same energy. Direct measurement of the output radiation at a heavily-shielded backwards angle isolates the NRF photons from the interrogating beam. The isolated NRF spectrum gives quantitative data about the composition of the target. However, NRF photons generated within the target will be heavily attenuated on the way out, resulting in low count rates at the detector.

Instead of measuring the NRF spectrum directly, we can also use a tomography-like approach to measure the effects the target has on the transmitted interrogation beam. This allows us to more accurately find the total amount of NRF absorption that occurred in the target. Since NRF radiation is emitted isotropically, the large majority of NRF photons will travel in a different direction than the original interrogating beam. A direct measurement of the transmitted beam shows how much of it was absorbed or scattered, thereby providing information on the nuclides present in the target. However, the interrogating beam itself is generally too intense for a direct measurement. Also, one is often concerned only with particular features of the output spectrum, i.e. at energies near the resonances of a specific nuclide.

A witness foil may be used to address the issues involved with direct measurement of the beam exiting the target. These foils are composed of a single type of nuclide (at least as pure as manufacturing allows), and therefore respond in a very well-defined manner to an interrogating beam. NRF in the foil is the dominant effect over an energy range comparable to the energy resolution in HPGe detectors (~3 keV). Therefore, one can expect significant differences in the emissions from the witness foil, depending on how the initial target affects the spectrum of the interrogation beam.

For example, consider two different targets and a test using a ^{235}U witness foil. One of the targets contains almost no ^{235}U, while the other is enriched to, say, 20%. A bremsstrahlung source creates a continuous photon spectrum with a large flux at 1.733 MeV (one of the excitation energies of ^{235}U). Unless ^{235}U is present in the target, very few of the 1.733 MeV photons in the beam interact within the target. Therefore, when this beam interacts with the low-enrichment target, it is mostly unattenuated at this key energy. When the beam then hits the witness foil, large NRF effects are observed. But in the high-enrichment target, a large fraction of the 1.733 MeV intensity is absorbed within the target. Thus, fewer of these photons remain to induce NRF within the foil. The emitted spectrum from the foil depends strongly on how much ^{235}U is present in the target.

THE PROBLEM

While it is clear that the NRF spectrum from a witness foil depends on the composition of the initial target, we need to know the relationship precisely to effectively scan nuclear material. Obtaining reliable analytical calculations is the first step towards this goal. Hand calculations for all but the simplest cases are untenable, due to the large number of nuclides involved and the complexity of cross-sections as functions of energy. MCNP seems the natural solution to process the large numbers of computations needed. However, we have recently uncovered flaws hardwired into MCNP's treatment of scattering physics. (See Brian Quiter's work on Rayleigh scattering for high-Z nuclei for a detailed description of the problem.) These flaws made MCNP unsuitable for our simulations.

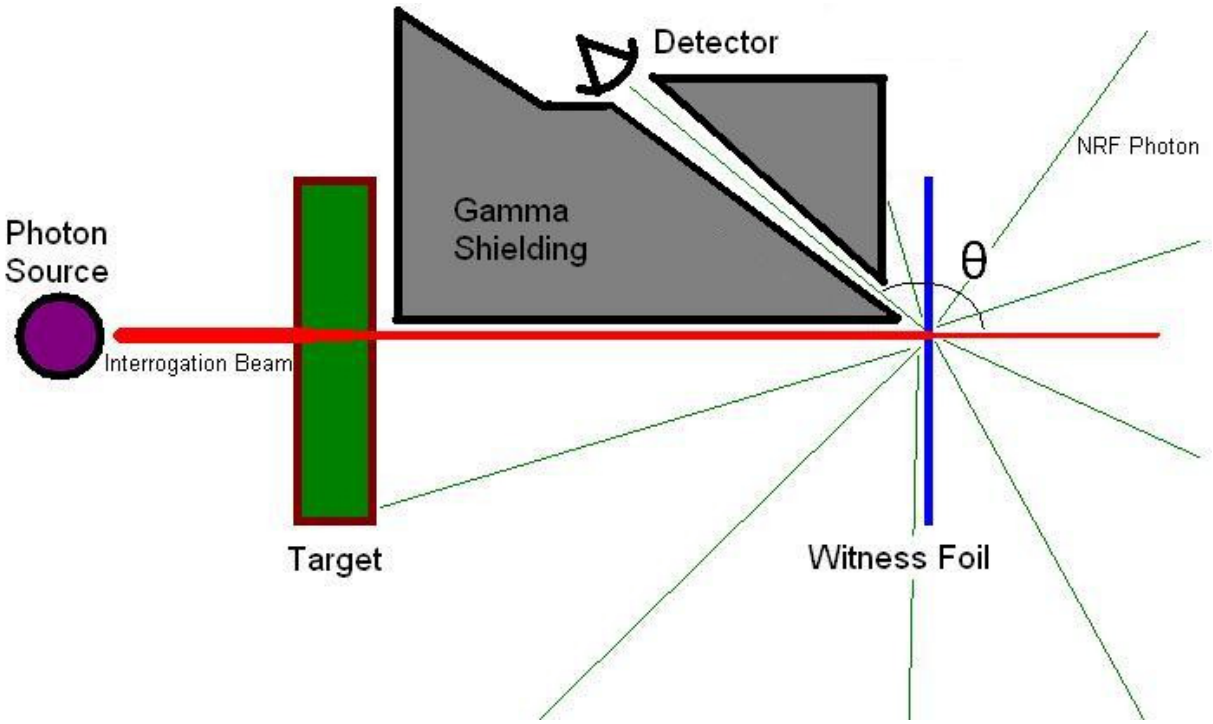

Figure 1: The problem geometry

THE SOLUTION

To work around these issues, we have written a set of Matlab tools designed to simulate a simple witness foil geometry. The code relies on user input for cross-section data; this allows one to include or remove various interactions at will by modifying the cross-sections. One must also specify the compositions of the target and the witness foil, among other physical parameters. A detailed description of the input can be found in the documentation section of this paper.

The toolset currently requires 6 Matlab functions to run:

- **witnessCalc**
  This is the main control interface. It reads cross-section data from an input file, calls methods to calculate attenuation coefficients, computes output spectra, and writes the results to a text file.
- **fileReader**
  Reads numerical data from a delimited text file into a matrix in Matlab. Note that the input file must be rectangular in that each line contains the same number of fields. The function skips any line containing non-numerical text.
- **parseTXTline**
  A helper method for fileReader, which reads a delimited text string and returns an array of values
- **atomToMassPercent**
  witnessCalc supports composition inputs in either relative atomic abundance or in mass percent. Since it is often easier to compute atomic abundances, this function converts the composition into mass percent form. This form is then directly used for calculating photon attenuation within both the target and foil.
- **attenuator**
  This function takes composition and cross-section data to compute attenuation coefficients as functions of energy.
- **writeDataToTxt**
  Matlab's data output is remarkably clunky and ill-suited for text files. This function allows one to more easily print data arrays directly to a specified text file.

In addition, there is a set of 4 methods to aid in the construction of the main cross-section data file (generally called masterArray.csv).

- **masterArrayCreator**
  A Matlab script that computes cross-sections for NRF as well as incoherent and coherent scattering. The script also writes all this data to the master cross-section file.
- **calc_XCOM_XS**
  Takes an input mass attenuation coefficient from the XCOM library and converts it to a cross-section in barns for a given nuclear mass.
- **calc_NRF_XS**
  Models the form of an NRF cross-section, given centroid energy, a FWHM for the distribution, and a list of energies. Three models are available: square, Gaussian, and pyramidal.
- interpolateLine
  A very simple method that takes two data points and approximates a value at a third point.

COMPUTATIONAL METHODS

To compute the intensity of photons emitted by the witness foil, we must find the attenuation of the interrogation beam through each material. Suppose that a nuclide n_i has a cross-section function $\sigma_i(E)$ and a number density of N_i atoms per unit volume. Then the attenuation coefficient as a function of energy is

$$\mu_i(E) = \sigma_i(E)\, N_i$$

The total attenuation coefficient for the material is $\mu(E) = \Sigma\, \mu_i(E)$ summed over all nuclides present. Then, given an incident photon intensity of $I_0(E)$, the attenuated intensity of the output beam is

$$I(E) = I_0(E)\, e^{-\mu(E)\, x}$$

Where x is the length of the path the beam follows through the material. For normal incidence, x is simply the thickness of the material.

Note that the number density of nuclide n_i can be calculated from its atomic mass and the density of the solid material it is part of. Specifically, if ρ is the density of the material, A_i is the atomic mass, and M_i is the mass percent of the nuclide within the material,

$$N_i = (\rho\, N_A\, M_i\, /100)/\, A_i$$

where $N_A = 6.022 \times 10^{23}$ is Avogadro's number. The atomic abundance M_i' is related to the mass percent M_i by the formula

$$M_i = A_i\, M_i' \,/\, M$$

Where $M = \Sigma\, A_i\, M_i'$ summed over all nuclides present.

For our purposes, we assume the following geometry. The interrogating beam is incident normal to the target, which has a thickness D1 and attenuation coefficients μ1(E). Therefore, the spectrum of photons leaving the target (and hence which are incident on the witness foil) is defined by

$$I_{\text{to foil}}(E) = I_0(E)\, e^{-\mu 1(E)\, D1}$$

This beam then interacts within the foil, producing NRF photons. Note that elastic scattering can also play a significant role in the detected signal. While elastic scattering cross-sections are generally very low compared to those for NRF, we will only see NRF across a narrow energy range (several eV). The energy resolution of an HPGe detector is on the order of 3 keV, so elastic scattering within 1500 keV of the resonance will contribute to the total detected peak. We may also have inelastic backscattering from higher-energy source photons, but this has not been accounted for in our programming yet.

Assume that the effects from NRF and elastic scattering can be linearly superposed. That is, let

$$I_{NRF}(E) = I_{\text{to foil}}(E)\, (1 - e^{-\mu 2(E)\, D2})$$

given a foil thickness D2 and attenuation coefficients μ2(E). Similarly, define the elastic scattering intensity as

$$I_{elastic}(E) = I_{\text{to foil}}(E)\, (1 - e^{-\mu 3(E)\, D2})$$

At this point it is essential to account for the solid angle Ω that the detector makes relative to the foil. Since NRF is emitted isotropically, we can multiply by $\Omega / 4\pi$ to get the fraction of NRF that heads toward the detector. For elastic scattering, which is decidedly not isotropic in nature, one must use the differential cross-section. One can easily factor in solid angle for elastic scattering by putting

$$\mu 3(E) = \Omega\, (d\sigma_i(E)/d\Omega)\, N_i$$

This approach assumes that the detector solid angle is small enough that the elastic scattering flux across the detector surface is constant.

The beam of photons that heads towards the detector has a total intensity of

$$\mathbf{I_{out}(E) = I_{NRF}(E) + I_{elastic}(E)}$$

However, this output spectrum is itself attenuated as it exits the foil towards the detector. Since the NRF and elastic emissions occur at different depths within the foil, we must find an "average" attenuation distance. Let θ be the angle from the original beam axis in the foil to the detector. Assume that the distance to this detector is large enough relative to the thickness of the foil that θ does not change throughout the foil. By this, assume that the attenuation depth as seen by the NRF photons is constant over the whole solid angle subtended by the detector.

Define the spectrum that exits through the back of the foil as $I_{transmitted}(E)$. Then, use this quantity to define the average intensity within the foil to be $I_{avg}(E) = (I_{to\ foil}(E) + I_{transmitted}(E))/2$. The effective attenuation coefficient due to the witness foil is

$$\mathbf{\mu_e(E) = \mu(E) (1 + |\cos \theta|^{-1})}$$

Let the average attenuation depth then be defined

$$\mathbf{D(E) = -\ \mu_e(E)^{-1} \ln (I_{avg}(E)/I_{to\ foil}(E))}$$

Finally, the NRF + elastic scattering intensity that emerges from the foil at an angle of θ is:

$$\mathbf{I_{final}(E) = I_{out}(E)\ e^{-\mu 2(E)\ D(E)}}$$

(Recall that μ2 is the attenuation coefficient for NRF photons within the foil.) This final intensity is the signal we expect to see near the resonance.

A NOTE ON CROSS-SECTIONS

The witnessCalc toolset requires manual input of cross section data in a comma-separated value (*.csv) file. The precise format of this file will be discussed below in the documentation section. For now, we will discuss the general methodology behind the cross-sections for each step of the calculation.

As explained in the previous section on computational methods, there are three general attenuation steps to consider. First, the interrogation beam interacts within a thick target material. The portion of the beam that exits this material then comes in contact with a witness foil. Part of the beam excites NRF emissions within the foil. Elastic scattering can also occur, but generally has much smaller probability at any given energy. Finally, the NRF and elastically-scattered photons exit the foil to the detector, and are attenuated while in the foil.

For the first interaction step, we want to find the total intensity of the beam passing through the target such that the photons will afterward be incident on the witness foil. Heavy shielding between the target and the foil collimates the beam, so that scattered photons do not reach the witness foil. Since photons absorbed within the target by NRF are re-emitted isotropically, the vast majority will not reach the foil either.

Note that elastic scattering is usually a negligible effect compared with either NRF or inelastic scattering. Therefore, the only interactions we count towards attenuation in the target are inelastic scattering and NRF. These are subtracted from the initial beam intensity.

The second interaction is not so much about beam attenuation as it is about NRF production. The photons produced within the foil are of interest only if they emerge with energy close to the NRF peak in question. (Consider that an HPGe detector has roughly 3 keV energy resolution.) Inelastically-scattered photons will generally fall well outside this range, especially for large backwards angles (where we place our detector). However, elastic scattering does not change a photon's energy. So, photons which elastically scatter towards the detector will register in the same energy bins as any NRF photons.

Thus, we combine the cross-sections for NRF and elastic scattering to determine the strength of the source within the witness foil.

The final attenuation step is very similar to the first. Shielding between the foil and the detector limits the angles that photons can emerge with in order to hit the detector itself. Thus, a photon that undergoes any scattering or NRF event or will most likely not reach the detector.

Again, elastic scattering is negligible compared to the other effects, and so the cross-sections for such events can be omitted from the total. The attenuation in the final step is due entirely to NRF absorption and inelastic scattering.

Now that we have addressed what cross sections should be used where, we will explain how the cross sections themselves can be obtained. All of the conversions and calculations detailed below have been implemented in Matlab helper functions, which can be called from masterArrayCreator. We used a combination of modeling and published libraries. Mass attenuation coefficients for photon scattering can be looked up in the XCOM database, located at http://physics.nist.gov/PhysRefData/Xcom/Text/XCOM.html. (DOS and Unix downloadable packages are

also available if you prefer to have the data on hand.) Be careful to use the data for *inelastic* scattering only! XCOM lists these coefficients as functions of energy, in units of cm^2/g. To use them with the witnessCalc toolset, they must be converted to barns. Let $(\mu/\rho) = M$ be the mass attenuation coefficient from XCOM (cm^2/g), while σ is the cross section, N is the number density ($atoms/cm^3$), A is the nuclear mass (g/mol), and G is Avogadro's number (atoms/mol). By definition, $\mu = \sigma N$. It is also clear that $\mu = \rho (\mu/\rho) = \rho M$. Therefore,

$$\sigma = \rho M / N = (A N / G) (M / N) = A M / G$$

This gives σ in cm^2, so a final multiplication by 10^{24} yields the cross section in barns. Note that the scattering cross-sections are very nearly constant over the small energy ranges typical of NRF peaks.

Elastic scattering cross-sections have been tabulated in the RTAB database for a variety of nuclides. We used the recently-computed S-matrix values released by Prof. Lynn Kissel. While the tabulations are well-populated for lower energies, there remain large gaps in the data for higher energies. In particular, RTAB does not contain information for scattering at 1.733 MeV. We wrote a set of Python scripts to linearly interpolate between entries in the database, which are included in the source code at the end of this document. Note that RTAB lists differential cross-sections.

Finally, we modeled NRF cross-sections using three models. The first, which is the physical model, gives a simple Gaussian due to Doppler-broadening. For ^{235}U, the 1733 keV resonance peak has a FWHM of 1.4 eV and integrates to 36 barn · eV, yielding a maximum value of 24.158 barns. The NRF curve can be modeled to as fine of resolution as desired; we used 0.1 eV in all our simulations. The second cross-section model is the easiest to work with by hand—it is zero for all E except in the region within FWHM/2 of the resonance peak, where it takes a constant value.

The third model is an approximation of the Gaussian form. Multiple evenly-spaced points are evaluated along the Gaussian curve, and then straight line segments are draw between them. In MCNP, the Gaussian form is approximated with a 5-point, two-step pyramid shape. The points are evaluated at E0 – 4, E0 – 2, E0, E0 + 2, and E0 + 4 eV, where E0 is the centroid energy in eV.

DOCUMENTATION

This section documents the use of witnessCalc, including input/output files and command lines. As mentioned in the introduction, this toolset includes 10 Matlab functions, 6 of which are required to run the chain properly. Below is a complete description of the command-line use for each function.

- **[postTarg_spec,final_spec] = witnessCalc(inputTable,atomicOrMass,targ_comp,targ_thick,targ_dense,foil_comp,foil_thick, foil_dense, detectorSolidAngle,detectorAngle,output_filename)**

  All parameters except output_filename are required. If output_filename is not given, no output file is written.

  - **inputTable** is a string that lists the name of main cross-section data file to be used. This file must be located in the current working directory in Matlab
  - **atomicOrMass** is a switch with two options: 'atomic' or 'mass'. This specifies if atomic abundance or mass percent was used to define compositions. Note that this option must be the same for both the foil and the target.
  - **targ_comp** is a three-column array detailing the composition of the target. It has the following format:
    - Column 1: 5-digit ZAID
    - Column 2: Relative composition (in atomic % or mass %)
    - Column 3: Column index in inputTable for cross-section data for the nuclide
  - **targ_thick** simply gives the path length (in cm) of the beam through the target
  - **targ_dense** is the mass density of the target material, in g/cm^3
  - **foil_comp** has similar form to targ_comp, but specifies the foil composition. Note that it has 5 columns, to account for the extra complexity of the interactions within the foil:
    - Column 1: 5-digit ZAID
    - Column 2: Relative composition (in atomic % or mass %)
    - Column 3: Column index in inputTable for NRF cross-section data for the nuclide
    - Column 4: Column index in inputTable for elastic scattering cross-section data
    - Column 5: Column index in inputTable for NRF + inelastic cross-section data
  - **foil_thick** is the thickness of the witness foil in cm
  - **foil_dense** is the mass density of the foil material, in g/cm^3
  - **detectorSolidAngle** is the solid angle subtended by the detector, in steradians
  - **detectorAngle** is the angle between the interrogating beam direction and the line from the witness foil to the detector
  - **output_filename** is a string specifying the name of a tab-delimited text file to write output data to

  Output values

  - **postTarg_spec** is the relative intensity ($I_{to\ foil}/I_0$) of the photon beam that leaves the back of the target and is thus incident on the target.
  - **final_spec** is the relative intensity of the NRF lines that exit the witness foil. It is given in (I_{final}/I_0) / steradian.

Both output values are given as column vectors. The output written to output_filename includes a 5-column array of the form:

- Column 1: Energy (eV)
- Column 2: Source Intensity (in #/cm^2/s)
- Column 3: postTarg_spec (normalized to source intensity)
- Column 4: final_spec (normalized to source intensity, per steradian)
- Column 5: Absolute NRF spectra leaving the foil (per steradian)

In addition, the compositions and physical parameters for the foil and target, and a copy of inputTable are included.

- **dataArray = fileReader(filename,spacingChar)**
  The filename parameter is required; spacingChar is optional and defaults to ',' if not specified.
  - **filename** is a string giving the name of a text file to read into Matlab's memory.
  - **spacingChar** represents the character used to delimit the text file. '\t' (tabs) and ',' (commas) are the most common.
  - **dataArray** is the Matlab matrix of numerical data as retrieved from the file in question.

  Note that inputTable must be a *.csv file within this toolchain. fileReader is called using the default comma delimiter.

- **[dataFields] = parseTXTline(line,spacingChar)**
  The filename parameter is required; spacingChar is optional and defaults to ',' if not specified.
  - **line** is a delimited text string to separate into a vector of fields
  - **spacingChar** represents the character used to delimit the text file.
  - **dataFields** is a Matlab vector of string data as retrieved from the text line in question.

- **massPerc = atomToMassPercent(atomPerc)**
  - **atomPerc** is a composition array, as used by witnessCalc. It contains 3 columns as follows:
    - Column 1: 5-digit ZAID
    - Column 2: Relative composition (in atomic %)
    - Column 3: Column index in inputTable for cross-section data for the nuclide
  - **massPerc** is the same composition array as atomPerc, except that column 2 now lists the composition in mass percents

- **coefficients = attenuator(composition,density,masterArray)**
  - **composition** is a three column array with the standard form used for compositions in this toolset. The columns are:
    - Column 1: 5-digit ZAID
    - Column 2: Relative composition (in mass %)
    - Column 3: Column index in inputTable for cross-section data for the nuclide
  - **density** is the material mass density of the material through which a beam is attenuated
  - **masterArray** is the numeric array of cross-sections as functions of energy, as read from **inputTable** in witnessCalc
  - **coefficients** is a vector of attenuation coefficients as a function of energy

- **writeDataToFile(filename,spacingChar,varargin)**
  - **filename** is a string that gives the name of the file to write output data to
  - **spacingChar** specifies the delimiting character to write the output with
  - **varargin** can be any number of cells, strings, or arrays to be written to an output file

  Note that newlines are automatically inserted between each dataset given in varargin. To insert special characters or extra delimiters, one must write them directly into the arrays.

The main cross-section data is taken from a comma-separated value (*.csv) file whose name is specified by the inputTable argument in witnessCalc. The structure of this file is as follows:

- Column 1: Energy in keV
- Column 2: Interrogation beam intensity, in #/cm^2/s
- Columns 3+: Cross-sections (in barns) for the nuclides used in the calculation

Below are descriptions for the tools that help create this *.csv file.

- **masterCell = masterArrayCreator()**
  - **masterCell** is a Matlab cell containing all the data in the main data file it creates.

  Note that all settings are specified within the Matlab file itself; it runs as a script. One must construct an array of energies to use, as well as an accompanying array for source strength at each energy.

  Furthermore, once the energy bins are set up, one must then make a list of nuclide cross-section data to add to masterCell. This generally happens in two steps: 1.) calculate the relevant cross-sections for the nuclide, 2.) write the data (with a header title) to masterCell.

  This method makes use of writeDataToFile.

- **XCOM_XS = calc_XCOM_XS(XCOM_mass_atten,A)**
  - **XCOM_mass_atten** is the mass attenuation coefficient (in cm^2/g) from the XCOM library
  - **A** is the nuclear mass in g/mol
  - **XCOM_XS** is the scalar value of the scattering cross-section in barns

  One should be careful to use the correct mass attenuation coefficient for this method. The "standard table" in XCOM does not have data for all energies, but one can specify additional energies for output. XCOM uses a log-log cubic spline for most of its interpolation. Also be sure to use incoherent attenuation only when working with nuclides within the target.

- **NRF_XS = calc_NRF_XS(centroid,FWHM,integrated_XS,mode,energies,N_pts)**
  - **centroid** marks the center of the NRF cross-section function in energy. It should be input as eV.
  - **FWHM** is the full-width-half-max (or simply width) of the distribution, in eV
  - **integrated_XS** gives the integrated cross-section of the NRF peak in eV·barns for normalization
  - **mode** is a string which determines which cross-section model to employ. Choices are "square", "gaussian", and "pyramidal". The square model is constant and nonzero within the width of the peak, but zero everywhere else. The Gaussian model is more correct physically, and yields a Gaussian cross-section distribution. Finally, the pyramidal model

peaks at the centroid, and consists of multiple line segments. Let E0 represent the centroid coordinate, and suppose f(E) is a Gaussian defined by FWHM and E0. MCNP models the Gaussian shape around the five points:
1. (E0-4, f(E0-4)) 2. (E0-2, f(E0-2)) 3. (E0, f(E0)) 4. (E0+2, f(E0+2)) 5. (E0+4, f(E0+4))
The four segments go between points 1-2, points 2-3, points 3-4, and points 4-5. The peak height is then adjusted to yield the correct integrated cross-section.
However, one can specify an arbitrary number of points to define the pyramid shape. (This is the N_pts parameter.)

- **energies** is an array of the energies (in eV) to evaluate the cross-section at
- **N_pts** is used in conjunction with the "pyramid" mode. N_pts specifies the number of points to be used when defining the approximate Gaussian. Note that N_pts can range from 3 up to the length of the "energies" array minus 2. If N_pts is not specified, then the default value is 5 (which is used internally by MCNP).
- **NRF_XS** is an array of the evaluated NRF cross-sections in barns at each energy specified in the energies array

- **y3 = interpolateLine(x1,y1,x2,y2,x3)**
  - **x1** and **x2** are the abscissas of the two points to linearly interpolate with
  - **y1** and **y2** are the ordinates of the two points to linearly interpolate with
  - **x3** is the abscissa of the point to approximate
  - **y3** is the value of the line which passes through (x1,y1) and (x2,y2) at x3

EXAMPLE USE

To illustrate the use of witnessCalc, here is an example computation. The inputs below are written in the form one would use in Matlab.

```
masterArrayCreator();

targ_comp =
        [8016,55.1,5;40090,17.2,4;92238,27.7,3];

foil_comp = [92235,100,6,7,8];

[postTarg_spec,final_spec] =
        witnessCalc('masterArray.csv','atomic',targ_comp,21.8,4,foil_comp,0.5,19.1,135,
        'witnessCalcData.txt');
```

Here, the target is composed of 55.1% ^{16}O, 17.2% ^{90}Zr, and 27.7% ^{238}U by atomic abundance. Similarly, the foil is 100% pure ^{235}U. The target is 21.8 cm thick and has density 4 g/cm^3. The foil is 5 mm thick and has a density of g/cm^3. The detector is at a backwards angle of 135˚. Finally, all output is written to a file called witnessCalcData.txt in the current working directory.

The compositions above state that the cross section data for ^{16}O can be found in column 7 of "masterArray.csv". Similarly, ^{90}Zr cross sections are in column 6, while those for ^{238}U and ^{235}U are in columns 4 and 5, respectively. Make sure that the commands in the masterArrayCreator script match these settings! One must specify the output file name and order the cross-section columns manually.

In addition, the masterArray.csv file might look like: (not necessarily physical)

Energy (eV),Intensity (#/cm^2/s),binWidth (eV),U-238 XS (b),U-235 XS (b),Zr-90 XS (b),O-16 XS (b)
1731500,1e9,2,2,2,2,0.1,4
1732996,9e8,2,2,2,2.5,0.1,4.5
1732998,9e8,2,2,2,5,0.1,7
1733000,9e8,2,2,2,10,0.1,12
1733002,9e8,2,2,2,5,0.1,7
1733004,9e8,2,2,2,2.5,0.1,4.5
1734500,8e8,2,2,2,2,0.1,4

In more readable form, this translates to:

Energy (eV)	Intensity (#/cm^2/s)	U-238 XS XCOM (b)	Zr-90 XS XCOM (b)	O-16 XS XCOM (b)	U-235 NRF (b)	U-235 RTAB (b/ster)	U-235 NRF + XCOM (b)
1731500	1.00E+09	2	2	2	2	0.1	4
1732996	9.00E+08	2	2	2	2.5	0.1	4.5
1732998	9.00E+08	2	2	2	5	0.1	7
1733000	9.00E+08	2	2	2	10	0.1	12
1733002	9.00E+08	2	2	2	5	0.1	7
1733004	9.00E+08	2	2	2	2.5	0.1	4.5
1734500	8.00E+08	2	2	2	2	0.1	4

Table 1: A sample masterArray.csv

This table shows that ^{235}U has an NRF peak of 10 barns centered at 1.733 MeV, where the initial beam intensity gives 9.00E+08 photons/cm^2/s. Note that in the first and last bins are much wider than the others. Be sure to account for this in the source strength, i.e. do not give the number of photons per eV. This allows one to weight the spectrum appropriately to deal with extremely fine energy resolution and bulk bins at the same time.

masterArrayCreator.m is a script, and must be tailored for each use. There are several points of input, detailed below. All other quantities in the master *.csv file are automatically generated.

centroid, the resonant energy of the NRF peak in eV (line 9)
E_res, the energy resolution of the detector in eV (line 10)
integrated_XS, the integrated cross-section in eV barns (line 11)
m, the mass of NRF nucleus in u (line 21)

There are three types of cross-sections of interest. One must provide certain information for each (i.e. any actual numbers in the examples below), generally from cross-section tables:

XCOM calculation (adds the XCOM cross-section for O-16 to masterCell)

```
XCOM_atten = 4.811E-02;
XS = calc_XCOM_XS(XCOM_atten,15.995);
masterCell{1,col_index} = 'O-16 XS (xcom)';
masterCell(2:end,col_index) = num2cell(XS);
col_index = col_index + 1;
```

RTAB calculation (adds the 118-degree RTAB cross-section for U-235 to masterCell)

```
XS = interpolateLine(115,1.03886E-04,120,1.02684E-04,118);
masterCell{1,col_index} = 'U-235 XS (RTAB - barns/steradian)';
masterCell(2:end,col_index) = num2cell(XS);
col_index = col_index + 1;
```

NRF calculation (constructs a Gaussian NRF cross-section for U-235 for masterCell)

```
XS = calc_NRF_XS(centroid,FWHM,integrated_XS,'gaussian',energies);
masterCell{1,col_index} = ['U-235 XS (Gaussian NRF: ' num2str(FWHM) ' eV FWHM)'];
masterCell(2:end,col_index) = num2cell(XS);
col_index = col_index + 1;
```

In the last line of the masterArrayCreator, one can also specify the filename to output to.

COMPUTATIONS AT THE 1.733 MeV NRF PEAK

The first set of computations from witnessCalc focus on the form of the NRF cross-section for ^{235}U at 1733 keV. We modeled three functions for this task:

- A finite rectangle, with a 25.7-barn height and 1.4 eV width
- A Gaussian with a 24.16-barn peak and 1.4 eV FWHM
- A pyramidal shape, made from four line segments. The segment endpoints were defined by the values of the Gaussian fit at 1729 keV, 1731 keV, 1733 keV, and 1735 keV. This yielded a peak of 17.88 barns

Note that all three functions integrate to 36 eV • barns.

These three models help illustrate the effects of approximations on the detected NRF spectra. We computed very different behavior for these functions over a range of target enrichments.

In our computations, the source had a strength of 10^{10} photons/(eV • s • cm^2)over the entire range of interest--specifically 1733 ± 1.5 keV (approximately the resolution of an HPGE detector). The target contained 27.7% uranium, 55.1% oxygen, and 17.2% Zr by atomic abundance. This is a much simplified composition representing LWR spent fuel in a zirconium storage matrix. The target itself was a uniform 21.8 cm thick with a density of 4.0 g/cm^3. We varied the composition of the target by varying the enrichment of ^{235}U, while assuming that the oxygen and zirconium retained their natural isotopic abundances. Note that ^{235}U and ^{238}U are the only isotopes of uranium we considered. We computed data for enrichments of 0%, 0.5%, 1%, 5%, 10%, 25%, 50%, and 100% ^{235}U.

The witness foil was composed of pure ^{235}U, with a density of 19.1 g/cm^3. We compared two thicknesses: 1 mm and 5 mm. Also note that we assumed a detector positioned at a backwards angle of 135°, subtending a solid angle of 0.0053 steradians.

A computation was carried out for every combination of target enrichment, NRF cross-section model, and foil thickness, resulting in 48 runs total. We plotted two sets of data from these runs:

- Total detected countrate as a function of target enrichment
- Ratio of NRF photons vs. elastically scattered photons incident on the detector, as a function of target enrichment

Both plots are normalized to the results for 0% target ^{235}U enrichment.

The detected countrate is defined as the total number of photons that are incident upon the detector; it is the sum of the “Absolute Detected Intensity (counts/s)” column given in a witnessCalc output file. As discussed in the “Notes on Cross Sections”, NRF is the dominant effect close to the NRF peak, though rarer elastically-scattered photons can significantly affect the detected rate over a 3 keV energy range. (For comparison, NRF only acts over a range of a few eV.)

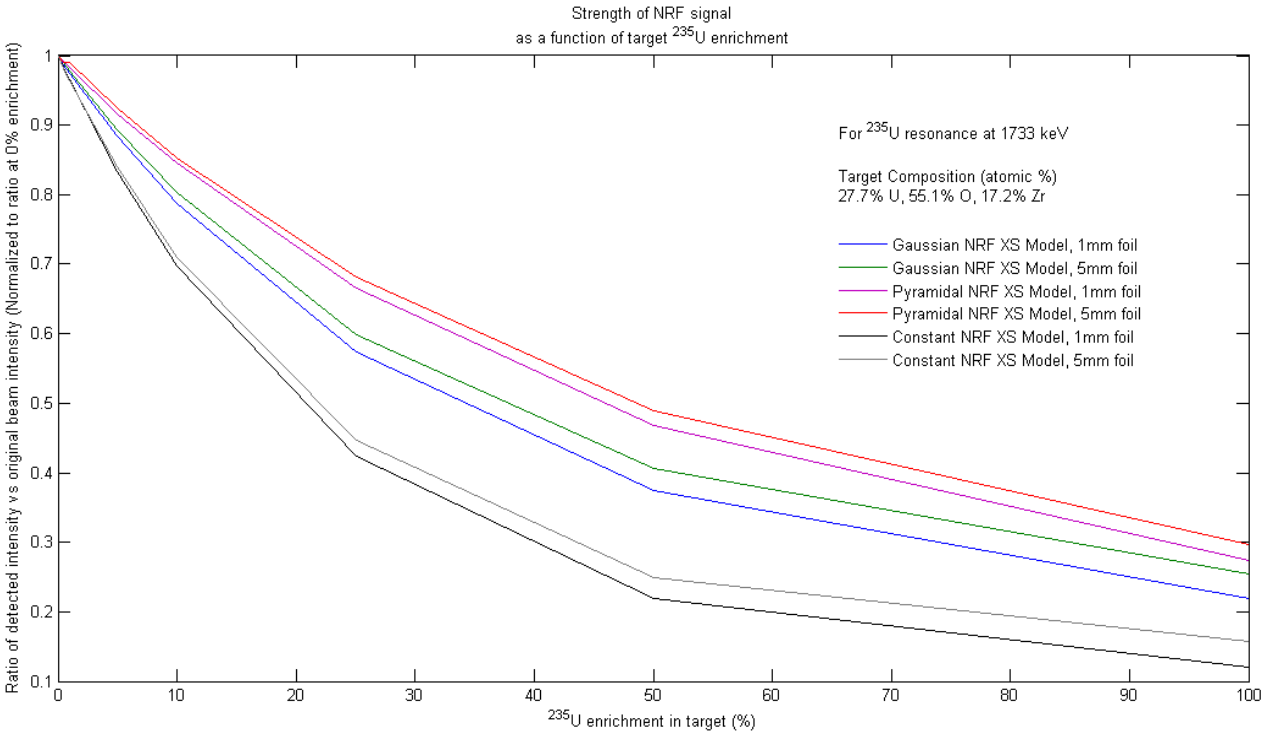

Figure 2: Detected count-rate as a function of target enrichment

Figure 2 shows a marked correlation between target composition and the number of resonant-energy photons incident upon the detector. In general, we note a roughly exponential decrease in the detected count rate as enrichment increases from 0% to 100%.

The six curves correspond to the different cross-section models for both 1 mm and 5 mm witness foil thicknesses. The thicker foil yields higher countrates within the detector, as expected. The very high concentration of ^{235}U in the foil makes NRF very much the dominant effect here. The count rates are not increased by a factor of 5, however, because of resonant attenuation within the foil itself.

Note that the square NRF cross-section model resulted in the lowest number of counts within the detector.

The NRF signal was on the order of 10^{-12} as strong as the original interrogation beam. For a source strength of 10^{10} photons/(eV • s • cm^2), we obtain a total detected yield of ~ 30,000 counts/s.

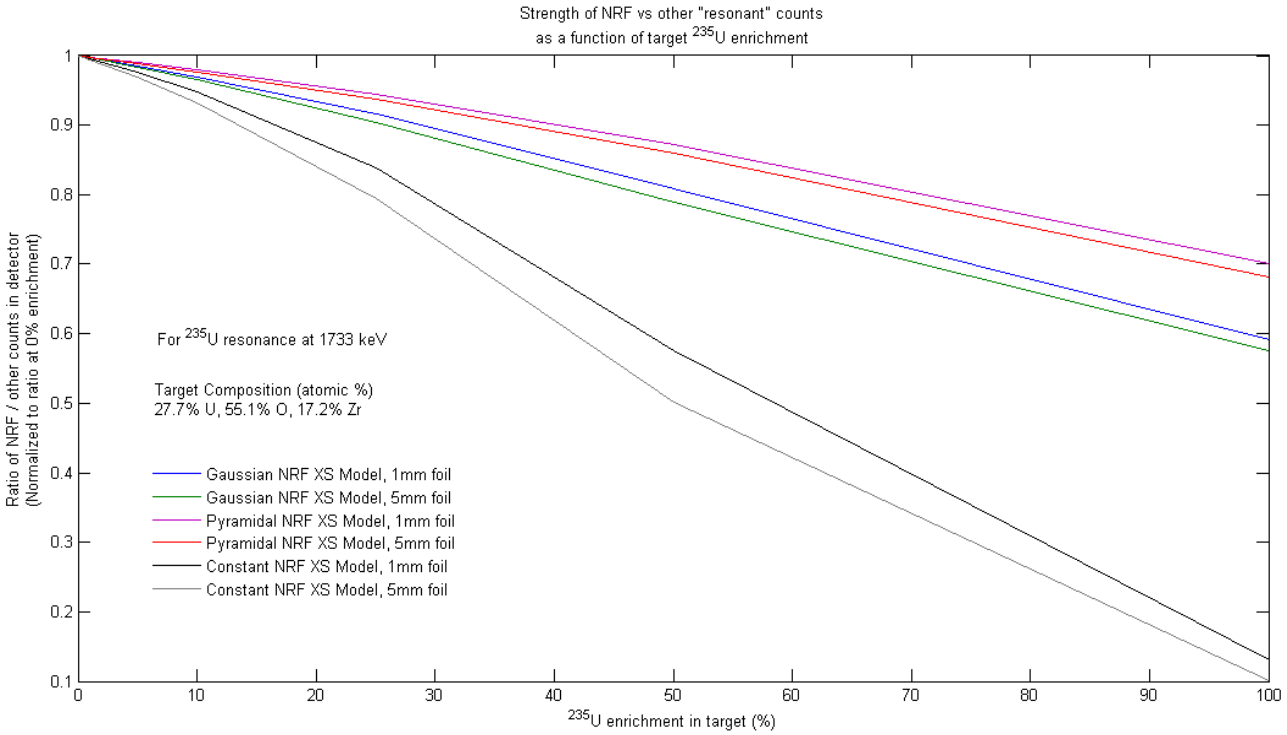

Figure 3: Ratio of NRF photons vs. elastically scattered photons incident on the detector

Figure 3 illustrates how elastic scattering becomes more significant for the output signal as target enrichment increases. This makes sense, noting that resonant photons tend to die off quickly due to a high absorption cross section. Therefore, far fewer photons within 1-2 eV of the NRF peak pass through the target to excite florescence within the foil. Since elastic scattering has a very small cross-section in comparison, photons within 1-2 keV of the peak generally reach the foil. These can then elastically scatter to the detector, contributing to the signal.

Note that the data for the square cross-section model drops off rather sharply compared to that from the other two models.

COMPARISON WITH EXPERIMENT: BRIAN QUITER AT MIT

In summer of 2009, Brian Quiter took a series of measurements at an NRF setup located at the Massachusetts Institute of Technology. The experiment focused on the resonances of ^{238}U at six energies: 2.176, 2.209, 2.245, 2.295, 2.410, and 2.468 MeV. Four targets of differing thicknesses and compositions served as a testbed. The first target contained no depleted uranium and acted as control for the experiment.

- Target 1: 7.60669 cm Pb
- Target 2: 7.00131 cm Pb and 0.44419 cm DU
- Target 3: 7.50531 cm Pb and 0.0882846 cm DU
- Target 4: 7.33631 cm Pb and 0.17538 cm DU

The witness foil was composed of pure depleted uranium, with a 0.407878 cm thickness. The depleted uranium contained 99.799% ^{238}U, with the balance roughly made up by ^{235}U. The figure in Appendix A shows a top-down view of the experimental geometry.

witnessCalc simulated the signal strength measured in the detector for each of the targets, using the three supported NRF cross-section models. A 5-point pyramid was used, to determine the magnitude of errors brought about by MCNP's approximation. The data was normalized to the signal strength generated from the control target (without DU). For all of the simulations, the NRF dominated the total detected signal. That is, the contribution of counts within 4 eV of the resonance peak accounted for 90% or more of the resolved signal at that resonance. (Elastic scattering contributed the remaining signal over the 3 keV resolution of the detector.)

Figure 4 plots 54 data points – each representing the normalized signal strength given a particular target, NRF model, and resonant energy.

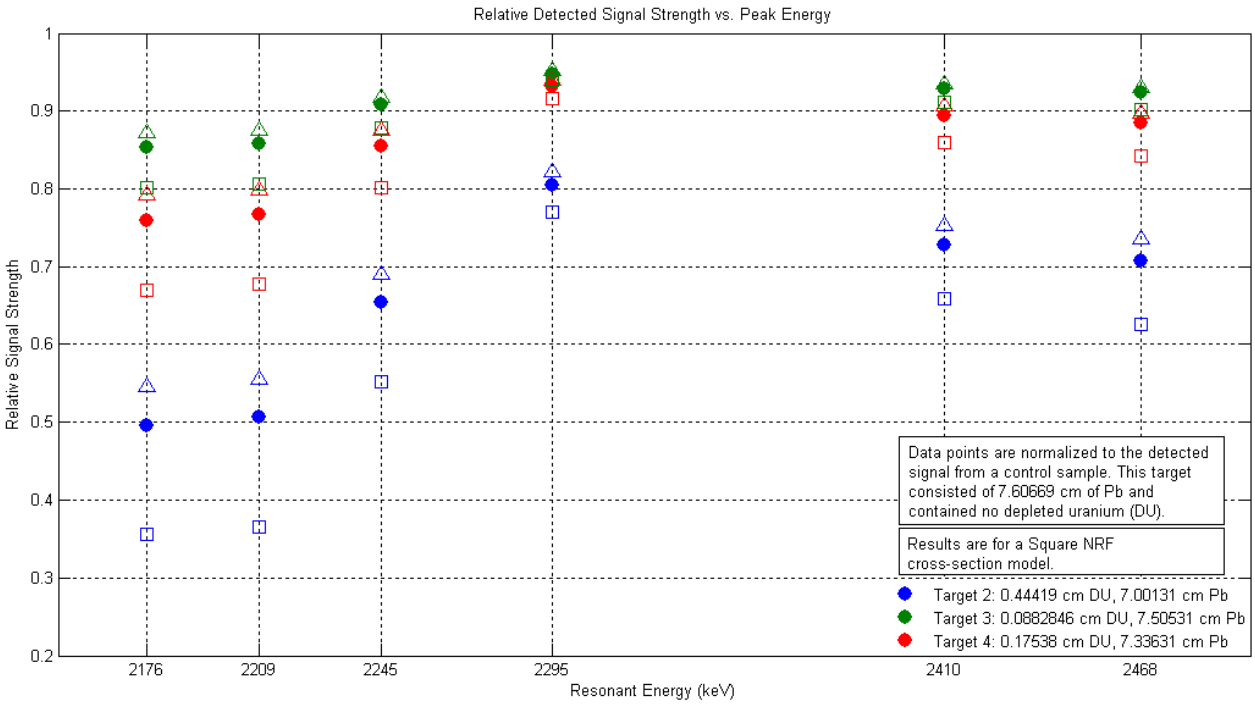

Figure 4: Normalized detected signal strengths

From the figure above, one observes that the square NRF cross-section model (denoted by squares) results in a consistently weaker signal than the theoretical (Gaussian) model (plotted as solid points). Furthermore, MCNP's pyramidal approximation (shown as triangles) can be as much as 10% too high. As expected, the detected signal becomes weaker as the amount of DU in the target increases.

A note about the simulations: the targets at MIT all consisted of several stacked plates of material. However, modeling multilayered plates is not feasible in this version of witnessCalc. Thus, we assumed that the targets were homogenous mixtures of DU and Pb.

A second set of runs determined statistical uncertainties due to the variance in the measured cross-sections for ^{238}U. In 1987, Heil et al measured six resonances in this nucleus. Their results are published in Heil et al, "Observation of Orbital Magnetic Dipole Strength in the Actinide Nuclei ^{232}Th and ^{238}U". Table 2 summarizes their findings.

Resonant Energy (MeV)	NRF cross-section, Average (eV · b)	NRF cross-section, -1σ (eV · b)	NRF cross-section, $+1\sigma$ (eV · b)
2.176	90.309	85.205	95.429
2.209	87.867	82.907	92.827
2.245	48.482	45.509	51.455
2.295	19.257	17.069	21.445
2.410	34.926	32.545	37.308
2.468	39.927	37.845	42.576

Table 2: Integrated NRF cross-sections for resonances of ^{238}U

We ran a series of simulations similar to the first set, but using only the Gaussian NRF cross-section model. This gave us information on the uncertainties in the detected signal strengths. Figure 5 (on the next page) gives a plot of signal strength with error bars, for each resonant energy. The plot contains data for each of the test targets.

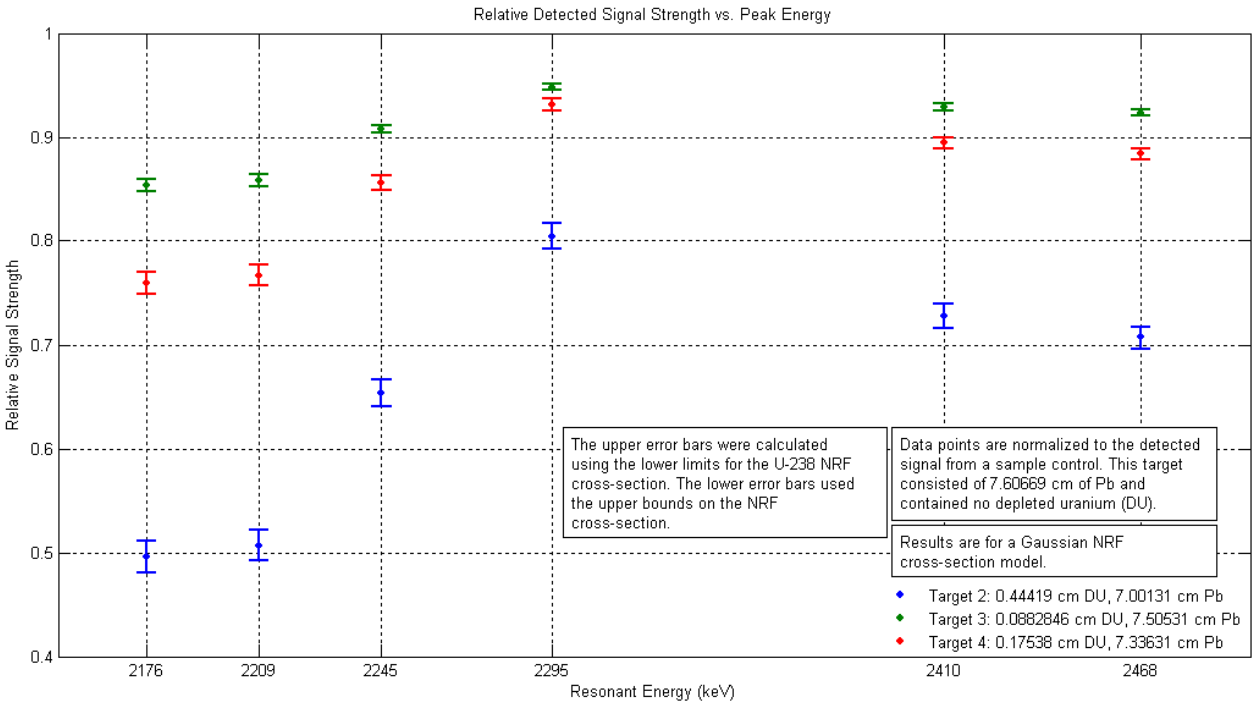

Figure 5: Uncertainties in signal strength, using the Gaussian NRF model

The uncertainties increase with target thickness, as is expected. NRF absorption effects are more pronounced the more material there is.

Next, we wanted to know how many points are required to accurately define a Gaussian cross-section. The definition of accuracy is somewhat arbitrary in this context. For our purposes, we defined an "accurate" Gaussian approximation as one which yielded normalized detected signal strengths within 1% of the "true" Gaussian shape. In this test, only the 2.176 MeV resonance was considered. Furthermore, only the control target (pure Pb) and target #2 were used.

Figure 7 shows the normalized signal strength for a number of approximations of the Gaussian NRF curve. Note that the "true" Gaussian model used was defined by an array of 81 points. Our simulations covered approximations using anywhere from 3 to 25 defining data points. (All the shapes integrated to the same total cross-section.) A few comparisons are given below in Figure 6.

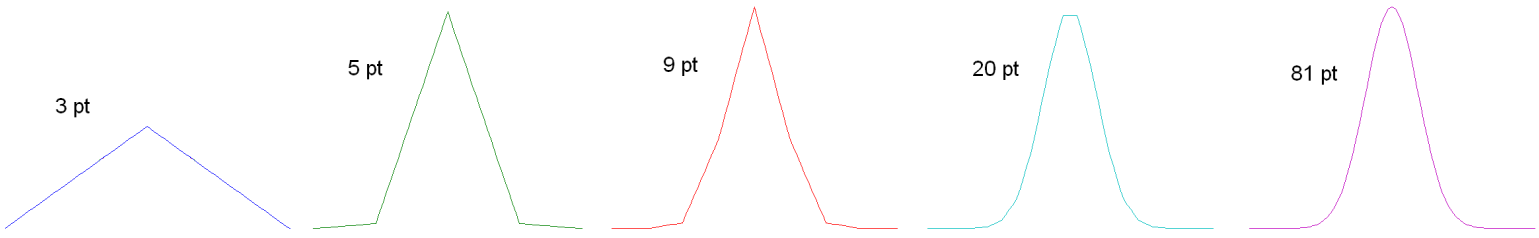

Figure 6: Various approximations of a Gaussian shape

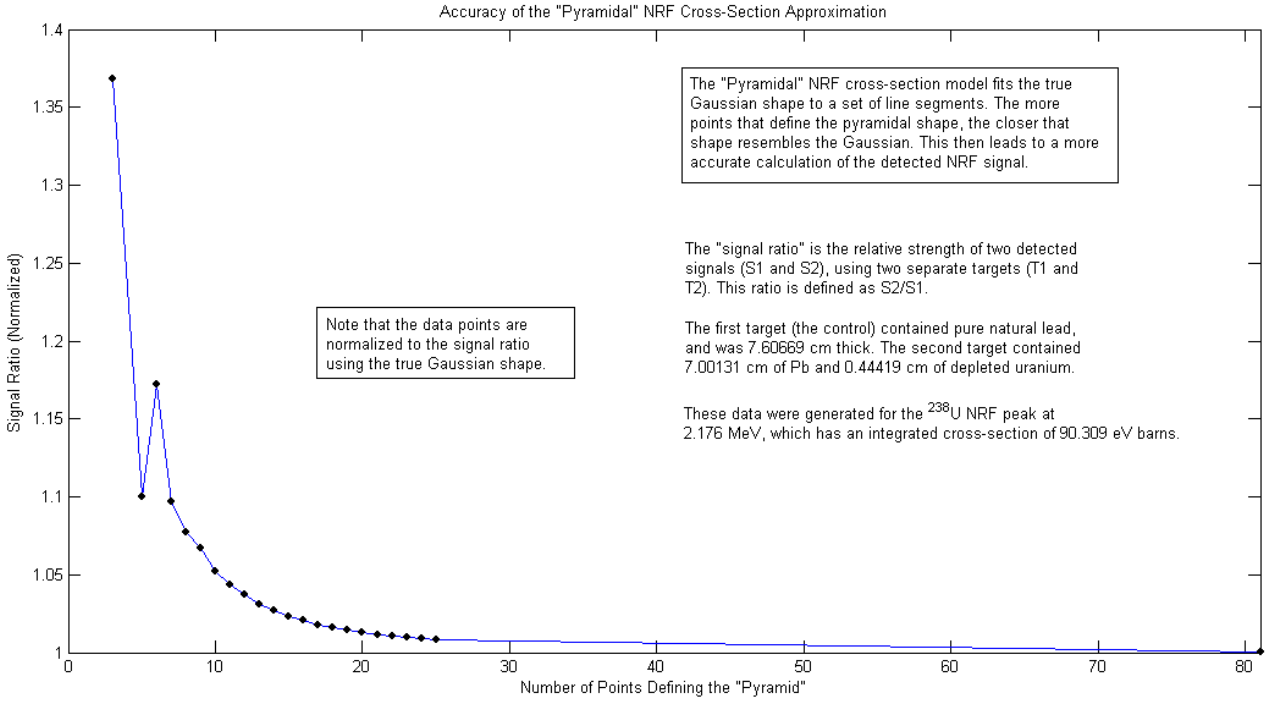

Figure 7: Number of anchor points required for an “accurate” Gaussian model

We concluded that it takes 23 defining anchor points to produce an “accurate” result. (Using 23 points, the detected signal strength was 0.98% above that for the 81-pt Gaussian.) MCNP uses only 5 anchor points, approximating the Gaussian shape as a two-step pyramid. This can lead to significant errors in the reported signal strength—on the order of 10% or more.

Further Work

Now that we have demonstrated a successful computational method in the witnessCalc toolchain, we can use it to simulate more complicated compositions. This will, of course, allow us to better predict results of upcoming experiments.

Our initial work used a single target thickness so as to study the effects of cross-section approximations. We have also compared results with experimental data from a few other target thicknesses, though they were all roughly the same. The main variable studied to this point is the composition of the target, i.e. the amount of target material present that will undergo NRF. The next step may be to vary the thickness of a target, while maintaining a constant amount of special nuclear material. There are far too many conceivable geometries and compositions to simply compute signature spectra for each. It would be far better to obtain an analytical expression predicting such spectra as functions of composition and target thickness.

We are also interested to find the foil thickness which maximizes our NRF countrate, which determines the precision with which we can identify the target.

REFERENCES

NIST XCOM – Photon Cross Section Database. National Institute of Standards and Technology, Physics Laboratory. http://physics.nist.gov/PhysRefData/Xcom/Text/XCOM.html

Heil, R. D. "Observation of Orbital Magnetic Dipole Strength in the Actinide Nuclei ^{232}Th and ^{238}U". Nuclear Physics A, Volume 476, Issue 1, 4 January 1988, Pages 39-47

Kissel, Lynn. RTAB: the Rayleigh scattering database. Radiation Physics and Chemistry, Volume 59, Issue 2, 1 August 2000, Pages 185-200

Matlab Online Documentation. The Mathworks. http://www.mathworks.com/access/helpdesk/help/helpdesk.html

Python Documentation. Python Software Foundation. http://www.python.org/doc/

APPENDIX A: MIT EXPERIMENTAL GEOMETRY

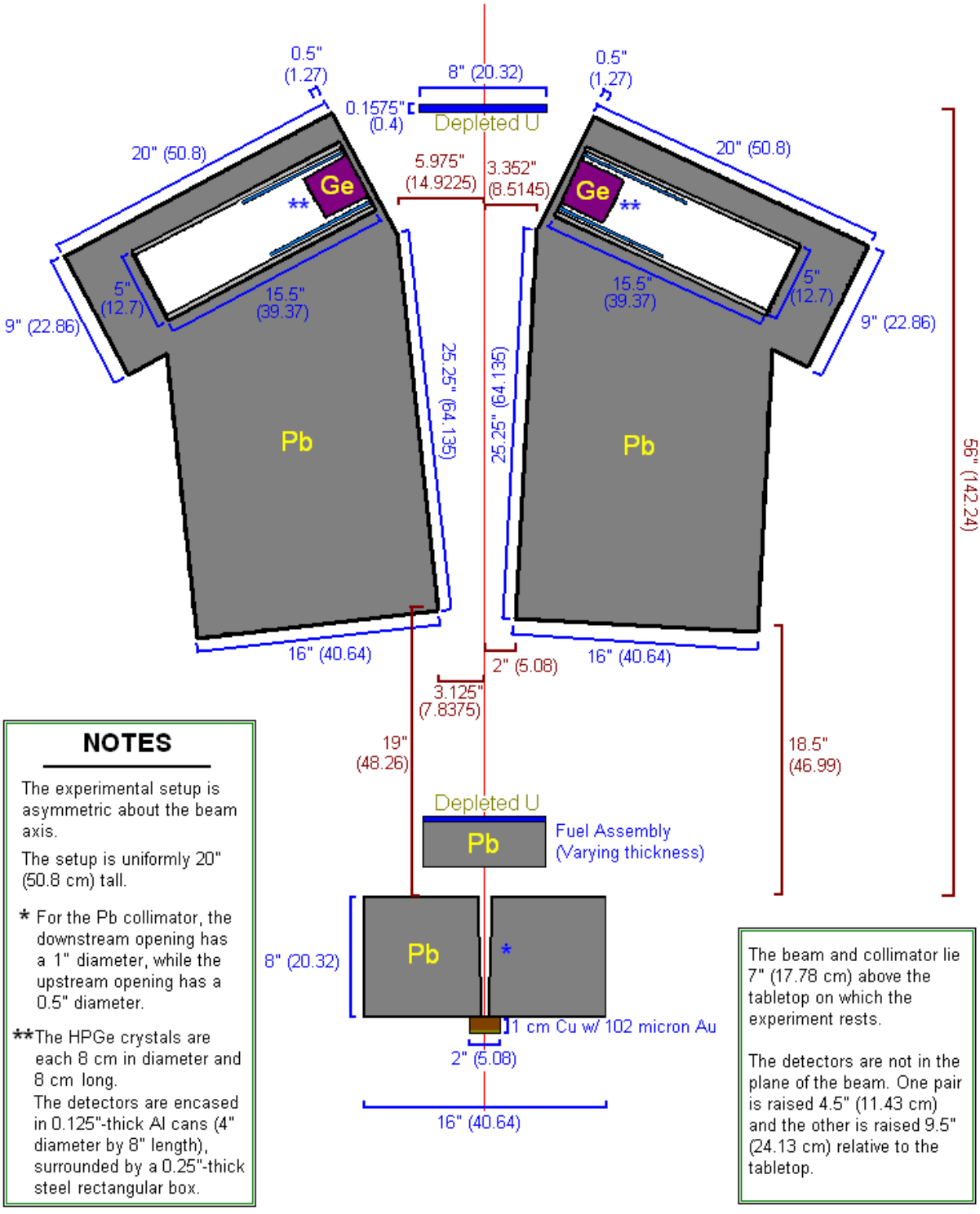

Figure 9: A detailed top-down view of Brian Quiter's witness foil experiment at MIT

APPENDIX B: SOURCE CODE

The source code for witnessCalc, fileReader, parseTXTline, atomToMassPercent, attenuator, and writeDataToTxt is given below, along with the scripts dealing with the RTAB database.

---

writeDataToFile.m

Allows one to more easily print data arrays directly to a specified text file

```
function writeDataToFile(filename,spacingChar,varargin)
% Function writes a tab-delimited file from data specified in varargin
% -- "filename" is the name of the file to be written
% -- "spacingChar" determines how the data fields in each object passed
%    through varargin should be spaced. For example, if spacingChar = '\t',
%    then the output file will be tab delimited.
% -- varargin contains any number of headers and/or data matrices to write
%    to a file. This function decides what each parameter is and prints it
%    appropriately.
fileID = fopen(filename,'wt');

% First, we determine how many objects there are to print
for i = [1:length(varargin)]
   data = varargin{i};
   % Matlab is very clunky for dealing with file output and strings
   % We need to convert each entry in data into a string to write
   % separately.
   [rows,columns] = size(varargin{i});
   for R =[1:rows]
      temp = cell(1,columns);
      for C = [1:columns]
         temp = data(R,C); % temp is the ith row of the dataArray matrix
         if isa(temp,'numeric') == 1
            temp = num2str(temp);
         elseif isa(temp,'cell') == 1
               temp = temp{1};
         else
            error(['Undefined data type. Cannot write as text']);
         end
         % Special characters will be interpreted literally. Let's
         % interpret them if they come up:
         if strcmp(temp,'\n')
            fprintf(fileID,'\n');
         elseif strcmp(temp,'\t')
            fprintf(fileID,'\t')
         % Print the data and a tab-delimiter, unless it's the last column
         elseif C == columns % Print a newline character instead of a tab here
            fprintf(fileID,'%s\n',temp);
         else
            fprintf(fileID,'%s\t',temp);
         end
      end
   end
end
fclose(fileID);
```

---

fileReader.m

Reads numerical data from a delimited text file into memory in Matlab

```
% Reads numerical data from a *.csv file into memory
% This function automatically removes any lines that contain non-number
% strings. Make sure the *.csv is rectangular and only contains one dataset!

function dataArray = fileReader(filename,spacingChar)
fileID = fopen(filename);
if fileID < 0
  error(['Could not open ',filename,' for input']);
end

% If spacingChar is not specified, this reads *.csv files by default
if nargin < 2
   spacingChar = ',';
end
% First we need to figure out the dimensions of our master array.
% "rows" represents the number of rows, while "cols" is the number of
% columns
status = 1;
rows = 0;
cols = 0;
while status > 0
   line = fgetl(fileID);
   % We only want to see how many fields are in the line
   junk = parseTXTline(line,spacingChar);
   tempCols = length(junk);
   % If this line has a different number of fields than the line before it,
   % we have a problem!
   if (tempCols ~= cols && cols ~= 0 && tempCols ~= 0)
      error(['*.csv data is not rectangular']);
   end
   if line == -1
      status = 0;
      break
   end
   rows = rows + 1;
   cols = tempCols;
end

% Need to reinstantiate the *.csv file for access
fileID = fopen(filename);

% strArray is a cell whose entries represent the columns in the *.csv file
% It is output as a cell of strings
strArray = cell(rows,cols);

% This loop populates strArray and then checks to see which rows have
% non-number strings in them
badRows = []; % Contains the row indices for the non-number strings
for i = [1:rows]
   line = fgetl(fileID);
   NaNFlag = 0;
   [data,junk] = strtok(line,';');
   % Now we parse the data by comma-delimiting
```

```
   fields = parseTXTline(data,spacingChar);
   for j = [1:cols]
      % Is the data point not a valid number?
      if size(str2num(fields{j})) == [0 0]
         NaNFlag = 1; % True if one or more entries is NaN
      end
      strArray{i,j} = fields{j};
   end
   % If we found a non-number, we'll skip the row for the final array
   if NaNFlag == 1
      badRows = [badRows i];
   end
end

% We have now read the *.csv file into memory. Now we want to take out
% header rows or any other rows containing text
[junk,rowsToDel] = size(badRows);
dataArray = zeros(rows-rowsToDel,cols);
index = 1;
for i = [1:rows]
   % Checking if badRows contains the current row index
   % Only writes good rows to dataArray
   if size(find(badRows == i)) ~= [1 1]
      for j = [1:cols]
         dataArray(index,j) = str2num(strArray{i,j});
      end
      index = index + 1;
   end
end

fclose all;
```

parseTXTline.m

A helper method that separates delimited text into an array of data

```
function [dataFields] = parseTXTline(line,spacingChar)
% Uses a while loop to extract data fields from a comma-delimited string

% If spacingChar is not specified, this reads *.csv files by default
if nargin < 2
   spacingChar = ',';
end

% Want to figure out how many data entries there are, first
% This loop finds the first field from the string, then loops using the line
% minus that field
counter = 0;
tempLine = line;
while length(tempLine) > 1
   [field,remainder] = strtok(tempLine,spacingChar);
   tempLine = remainder;
   counter = counter + 1;
end

dataFields = cell(1,counter);
% Must reinitialize the line to read into dataArray
tempLine = line;
for i = 1:counter
   [field,remainder] = strtok(tempLine,spacingChar);
   tempLine = remainder;
   dataFields{i} = field;
end
```

atomToMassPercent.m

Converts compositions from atomic abundances to mass percents

```
function massPerc = atomToMassPercent(atomPerc)
% Converting composition arrays from atomic percent to mass percent

% This function converts the material composition given by the
% atomPerc matrix, specified in atomic percentages, into the
% masses matrix. The output lists the composition in terms of mass
% percents.

% The input matrix should have two columns: the first gives ZAIDs for the
% component nuclides, while the second gives the atomic percentage within
% the material in question.

[nuclides,columns] = size(atomPerc);

% massPerc has an identical structure to atomPerc. Only column 2 is
% different
massPerc = atomPerc;
masses = zeros(nuclides,1);

totalAtomPercent = 0; % A quick check to make sure the composition was normalized correctly
for i = [1:nuclides]
   % What is the number density of the nuclide?
   ZAID = atomPerc(i,1);

   Z = double(uint16(ZAID/1000));
   A = ZAID - Z*1000;
   atomPercent = atomPerc(i,2);

   totalAtomPercent = totalAtomPercent + atomPercent;

   % Now to find the weighted mass proportions
   masses(i) = atomPercent*A;
end

% The total mass is the sum of all entries in masses; this lets us
% compute mass percents
totalMass = sum(masses);
massPerc(:,2) = 100*masses/totalMass;

if totalAtomPercent ~= 100
   sprintf('Warning: An atomic composition is not normalized to 100 percent!\nIt sums to %f percent.',totalAtomPercent)
end
```

attenuator.m

Computes attenuation coefficients from given compositions and cross-sections

```
function coefficients = attenuator(composition,density,masterArray)
% Used in tandem with witnessCalc to compute attenuation coefficients

% The input requires the composition, density, and thickness of the
% material through which a beam is attenuated. In addition, a masterArray
% contains all the cross-section data for the nuclides specified in the
% composition matrix.

%
% -- composition is a three-column matrix specifying the composition of the
%    target. The first column should list ZAIDs while the second lists the
%    relative abundance of nuclides (in mass %). The third column points to
%    the column in inputTable that contains cross-section data for the
%    nuclide.
% -- density is the density of the target in g/cc

% masterArray has the following structure:
% Column 1: Bin energy in keV
% Column 2: Intensity of the source as a function of energy within each bin
%          (in units of #/s/cm^2/eV for normalization)
% Column 3: Widths of the energy bins in column 2 (in eV)
% Columns 4+: Cross-sections of nuclides in the target or foil as a
%           function of energy (in barns)

[N_bins,columns] = size(masterArray); % Looks up how many energy bins there are

% Need to find the size of targ_comp to know how many nuclides are present
[nuclides,columns] = size(composition);

% The number of interactions in the target as a function of E, per unit time
coefficients = zeros(N_bins,1);

totalMassPercent = 0; % A quick check to make sure the composition was input correctly
for i = [1:nuclides]
   % What is the number density of the nuclide?
   ZAID = composition(i,1);
   Z = double(uint16(ZAID/1000));
   A = ZAID - Z*1000;

   massPercent = composition(i,2);

   N_density = (6.022*10^23)*(density*massPercent/100)/A;

   % Where is the cross-section data in masterArray?
   xs_col = composition(i,3);
   nuclide_xs = 10^-24*masterArray(:,xs_col);

   % Each individual component of the target contributes to the effective
   % cross section, and hence the attenuation coefficient:
   coefficients = coefficients + N_density*nuclide_xs;
end
```

witnessCalc.m

Main control method, computes final spectra and writes output files

```
function [postTarg_spec,detected_spec] =
witnessCalc(inputTable,atomicOrMass,targ_comp,targ_thick,targ_dense,foil_comp,foil_thick,foil_dense,detectorSolidAngle,
        detectorAngle,output_filename)
% Function for calculating the response spectrum from a witness foil

% A gamma source (whose distribution is defined in inputTable) is normally incident
% upon a target that may or may not interact with the beam via NRF. The
% photons which do not interact in the target then hit a witness foil,
% which gives off a spectrum which depends on the composition of the target

% -- inputTable is the filename of a *.csv file containing a gamma source
%    distribution. Structure is detailed below
% -- atomicOrMass specifies if input compositions list data in atom vs mass
%    percents. It can have two values: 'atomic' or 'mass'
% -- targ_comp is a three-column matrix specifying the composition of the
%    target. The first column should list ZAIDs while the second lists the
%    relative abundance of nuclides (in atom or mass %). The third column
%    points to the column in inputTable that contains cross-section data
%    for the nuclide.
% -- targ_thick is the thickness of the target material in cm.
% -- targ_dense is the density of the target in g/cc
% -- foil_comp is the same as targ_comp, but gives information about the
%    foil instead of the target
% -- foil_thick is the thickness of the witness foil material in cm.
% -- foil_dense is the density of the foil in g/cc
% -- detectorSolidAngle is the solid angle subtended by the detector, in
%    steradians
% -- detectorAngle is the backward angle at which the detector sits
%    relative to the foil
% -- If output_filename is given, witnessCalc will write the post-target
%    and final NRF spectra to that file in tab-delimited format. Otherwise,
%    no output file will be written.

% inputTable has the following structure:
% Column 1: Bin energy in keV
% Column 2: Intensity of the source as a function of energy within each bin
%         (in units of #/s/cm^2)
% Columns 3+: Cross-sections of nuclides in the target or foil as a
%           function of energy (in barns)

% Need a copy of csvReader for this to work
masterArray = fileReader(inputTable);

srcEnergies = masterArray(:,1); % Source spectrum particle energies
srcIntensity = masterArray(:,2); % Source spectrum strength: counts/s/cm^2

if strcmpi(atomicOrMass,'atomic') == 0 && strcmpi(atomicOrMass,'mass') == 0
   error(['Specify whether compositions are given in atomic percents or mass percents!'])
else if strcmpi(atomicOrMass,'atomic') == 1
      targ_comp = atomToMassPercent(targ_comp);
      foil_comp = atomToMassPercent(foil_comp);
   end
end
```

```
% Transport of the source beam will happen in three steps:

% First, the source spectrum is attentuated due to the target
targ_atten_coeffs = attenuator(targ_comp,targ_dense,masterArray);
postTarg_spec = srcIntensity.*exp(-targ_atten_coeffs*targ_thick);

% Next, the beam that goes through the target unscattered will be incident
% on the witness foil. The NRF spectrum produced within the foil is equal
% to the total number of NRF and elastic scattering interactions.
% We need to use different cross-sections for NRF production versus the
% attenuation of the NRF spectra on its way out the foil.

% foil_comp should have five columns. Columns 3-5 point to columns in
% masterArray that contain cross-section data. Column 3 is solely for NRF,
% column 4 is for elastic scattering, and column 5 is for all interactions
% that attenuate the NRF photons in the foil (i.e. NRF + inelastic).

NRF_comp = foil_comp(:,1:3);

% Defining the composition array for elastic scattering
% This array is the same as NRF_comp except for the last column
elastic_comp = NRF_comp;
elastic_comp(:,3) = foil_comp(:,4);

NRF_prod_coeffs = attenuator(NRF_comp,foil_dense,masterArray);
initial_NRF_spec = postTarg_spec - postTarg_spec.*exp(-NRF_prod_coeffs*foil_thick);
% Must factor in the solid angle of the detector. Assume NRF radiates
% isotropically.
initial_NRF_spec = initial_NRF_spec * detectorSolidAngle / (4*pi);

% To factor in solid angle for elastic scattering, we multiply the elastic
% scattering cross-section (given in barns per steradian) by the solid
% angle of the detector. attenuation coefficients depend linearly on
% cross-section
elastic_coeffs = attenuator(elastic_comp,foil_dense,masterArray)*detectorSolidAngle;
elastic_spec = postTarg_spec - postTarg_spec.*exp(-elastic_coeffs*foil_thick);

% The final attenuation step is through the foil to the detector. The foil
% attentuates the NRF_spec on its way out. We assume the thinness of the
% foil is sufficiently small compared to the distance to the detector that
% it does not affect the scattering angle.
% The interaction depth is also not constant within the foil; we'll need to
% calculate an average attenuation depth to determine what distance to
% attenuate over. Define the average *interaction* distance to be the depth
% for which the attenuated spectrum is halfway between the input and the
% output through the back of the foil. The average *attenuation* depth
% also accounts for the backwards-scattered attenuation of the NRF spectrum
% through the foil.
% Note the dependence of attenuation depth on photon energy.

% The total spectrum that heads in the direction of the detector:
NRF_elastic_spec = initial_NRF_spec + elastic_spec;

% The spectrum that exits the back of the foil without interaction
% We multiply by the solid angle again to account for ALL the NRF
% interactions, not just the ones that head towards the detector
postFoil_spec = postTarg_spec - (initial_NRF_spec*4*pi/detectorSolidAngle + elastic_spec);
```

```
avgIntensity = (postFoil_spec + postTarg_spec)./(2*postTarg_spec);
detected_spec = zeros(length(NRF_elastic_spec),1); % The spectrum that will be incident on the detector

% Defining the composition array for use in the final attenuation step
% This array is the same as NRF_comp except for the last column, which
% specifies cross-sections for all interactions that will attenuate the
% NRF on its way out
foil_atten_comp = NRF_comp;
foil_atten_comp(:,3) = foil_comp(:,5);

foil_atten_coeffs = attenuator(foil_atten_comp,foil_dense,masterArray);

for i=[1:length(avgIntensity)]
   if foil_atten_coeffs(i) ~= 0
      attenuation = foil_atten_coeffs(i)*(1+1/abs(cos(detectorAngle)));
      avg_depth = -log(avgIntensity(i))/attenuation;
      % avg_depth is the depth within the foil at which the intensity is
      % halfway between what was incident on the foil and what came out
   else avg_depth = log(2)*foil_thick;
   end
   detected_spec(i) = NRF_elastic_spec(i)*exp(-foil_atten_coeffs(i)*avg_depth);
end

% Lastly, divide by the source intensity to get the final spectrum due
% to NRF, independent of source. Dividing by 4*pi yields the NRF spectrum
% per steradian, assuming the detector solid angle is small enough that the
% attenuation depth within the foil is relatively constant.
% This also assumes isotropic emission.
postTarg_spec = postTarg_spec./srcIntensity;
detected_spec = detected_spec./srcIntensity;

% File output of spectra and settings if a filename is specified
if nargin > 10
   spec_header = {'Energy (eV)','Source Intensity (#/cm^2/s)','Normalized Post-Target Intensity','Normalized Detected
         Intensity','Absolute Detected Intensity (counts/s)'};
   output = [masterArray(:,1:2) postTarg_spec detected_spec detected_spec.*srcIntensity];
   detector_header = {'\n','Detector Solid Angle (steradians):',num2str(detectorSolidAngle),'\n','Detector Angle
         (degrees):',num2str(detectorAngle)};
   targ_header = {'\n','Target Composition','\n','Thickness (cm):',num2str(targ_thick),'\n','Density
         (g/cm^3):',num2str(targ_dense)};
   targ_header2 = {'\n','ZAID','Mass Percent','XS Column'};
   foil_header = {'\n','Foil Composition','\n','Thickness (cm):',num2str(foil_thick),'\n','Density (g/cm^3):',num2str(foil_dense)};
   foil_header2 = {'\n','ZAID','Mass Percent','NRF XS Column','Elastic Scattering XS Column','Attenuation XS'};
   masterArray_header = {'\n','Master XS Array Used'};

writeDataToFile(output_filename,'\t',spec_header,output,detector_header,targ_header,targ_header2,targ_comp,foil_header,
         foil_header2,foil_comp,masterArray_header,masterArray);
end
```

masterArrayCreator.m

Automates much of the work required to put an input cross-section file together

Note that this is just an example for a specific problem!

```
% Script for creating the masterArray.csv file used in witnessCalc
% Also returns the cell (masterCell) which will be written to the
% masterArray.csv file
function masterCell = masterArrayCreator()
% To create the array, we need to compute various cross-sections for
% the nuclides in both the target and the witness foil.

% Some general information about the NRF peak
centroid = 1733000; % Resonant energy of the NRF peak in eV
E_res = 1500; % Energy resolution of the detector in eV
integrated_XS = 36; % Integrated cross-section in eV barns

% We can calculate the width of the Doppler broadened NRF peak:
% FWHM = (2E/c)*(2 ln 2 * kT/m)^0.5
% where E is the energy of the resonance, T is the temperature (can
% generally assume T = 298K), and m is the nuclear mass
c = 2.998 * 10^8; % Speed of light in m/s
k = 1.381 * 10^-23; % Boltzmann's constant in (m^2 kg)/(s^2 K^1)
T = 298; % Temperature in K
u = 1.661 * 10^-27; % Mass of 1 au in kg
m = 235.051; % Mass of the NRF nucleus in u
FWHM = (2*centroid/c)*(2*log(2)*k*T/(m*u))^0.5; % Width of peak in eV

% This will be passed into the first three columns of masterCell below
energies = [centroid - E_res,(centroid-4):0.1:(centroid+4),centroid + E_res];
source = 10^10*ones(1,length(energies));

% Let's first create a blank cell that will eventually be populated and
% written to masterArray.csv

% The number of rows in masterCell is equal to the number of energies
% specified in the above array, plus 1 row for a text header
% We do not need to know the number of columns beforehand, but it
% can speed things up some if we can specify it here. Otherwise Matlab
% will dynamically recreate masterCell every time we write a new column
masterCell = cell(length(energies)+1,1);

% The first two columns in masterCell give information about the
% energy bins
masterCell(1,1:2) = {'Energy (eV)','Source Intensity (#/cm^2/s)'};
masterCell(2:end,1) = num2cell(energies);
masterCell(2:end,2) = num2cell(source);

% Next, we deal with the nuclide cross-section data
% There are three general steps per nuclide:
%   1. Lookup XCOM cross-section
%   2. Add in NRF if applicable
%   3. Write the header and XS array to a column in masterCell

% Note that the "standard table" from XCOM generally does not have
% attenuation coefficients for the energy in question. One must specify an
```

```
% "additional energy" for output. Also note that these cross-sections do
% not change appreciably over small energy ranges.

col_index = 3;

% XCOM mass attenuation for Oxygen
XCOM_atten = 4.811E-02;

% O-16 %
XS = calc_XCOM_XS(XCOM_atten,15.995);
masterCell{1,col_index} = 'O-16 XS (xcom)';
masterCell(2:end,col_index) = num2cell(XS);
col_index = col_index + 1;

% O-17 %
XS = calc_XCOM_XS(XCOM_atten,16.999);
masterCell{1,col_index} = 'O-17 XS (xcom)';
masterCell(2:end,col_index) = num2cell(XS);
col_index = col_index + 1;

% O-18 %
XS = calc_XCOM_XS(XCOM_atten,17.999);
masterCell{1,col_index} = 'O-18 XS (xcom)';
masterCell(2:end,col_index) = num2cell(XS);
col_index = col_index + 1;

% XCOM mass attenuation for Zirconium
XCOM_atten = 4.370E-02;

% Zr-90 %
XS = calc_XCOM_XS(XCOM_atten,89.905);
masterCell{1,col_index} = 'Zr-90 XS (xcom)';
masterCell(2:end,col_index) = num2cell(XS);
col_index = col_index + 1;

% Zr-91 %
XS = calc_XCOM_XS(XCOM_atten,90.906);
masterCell{1,col_index} = 'Zr-91 XS (xcom)';
masterCell(2:end,col_index) = num2cell(XS);
col_index = col_index + 1;

% Zr-92 %
XS = calc_XCOM_XS(XCOM_atten,91.905);
masterCell{1,col_index} = 'Zr-92 XS (xcom)';
masterCell(2:end,col_index) = num2cell(XS);
col_index = col_index + 1;

% Zr-94 %
XS = calc_XCOM_XS(XCOM_atten,93.906);
masterCell{1,col_index} = 'Zr-94 XS (xcom)';
masterCell(2:end,col_index) = num2cell(XS);
col_index = col_index + 1;

% Zr-96 %
XS = calc_XCOM_XS(XCOM_atten,96.908);
masterCell{1,col_index} = 'Zr-96 XS (xcom)';
masterCell(2:end,col_index) = num2cell(XS);
col_index = col_index + 1;
```

```
% XCOM mass attenuation for Uranium
XCOM_atten = 5.051E-02;

% U-238 %
XS = calc_XCOM_XS(XCOM_atten,238.051);
masterCell{1,col_index} = 'U-238 XS (xcom)';
masterCell(2:end,col_index) = num2cell(XS);
col_index = col_index + 1;

% U-235 (RTAB) %
XS = interpolateLine(115,1.03886E-04,120,1.02684E-04,118);
masterCell{1,col_index} = 'U-235 XS (RTAB - barns/steradian)';
masterCell(2:end,col_index) = num2cell(XS);
col_index = col_index + 1;

% U-235 (Gaussian NRF only) %
XS = calc_NRF_XS(centroid,FWHM,integrated_XS,'gaussian',energies);
masterCell{1,col_index} = ['U-235 XS (Gaussian NRF: ' num2str(FWHM) ' eV FWHM)'];
masterCell(2:end,col_index) = num2cell(XS);
col_index = col_index + 1;

% U-235 (XCOM + Gaussian NRF) %
XS = calc_XCOM_XS(XCOM_atten,235.042);
XS = XS + calc_NRF_XS(centroid,FWHM,integrated_XS,'gaussian',energies);
masterCell{1,col_index} = 'U-235 XS (xcom + Gaussian NRF)';
masterCell(2:end,col_index) = num2cell(XS);
col_index = col_index + 1;

% U-235 (Square NRF only) %
XS = calc_NRF_XS(centroid,FWHM,integrated_XS,'square',energies);
masterCell{1,col_index} = ['U-235 XS (Square NRF: ' num2str(FWHM) ' eV FWHM)'];
masterCell(2:end,col_index) = num2cell(XS);
col_index = col_index + 1;

% U-235 (XCOM + Square NRF) %
XS = calc_XCOM_XS(XCOM_atten,235.042);
XS = XS + calc_NRF_XS(centroid,FWHM,integrated_XS,'square',energies);
masterCell{1,col_index} = 'U-235 XS (xcom + Square NRF)';
masterCell(2:end,col_index) = num2cell(XS);
col_index = col_index + 1;

% U-235 (Pyramid NRF only) %
XS = calc_NRF_XS(centroid,FWHM,integrated_XS,'pyramid',energies);
masterCell{1,col_index} = ['U-235 XS (Pyramid NRF: ' num2str(FWHM) ' eV FWHM)'];
masterCell(2:end,col_index) = num2cell(XS);
col_index = col_index + 1;

% U-235 (XCOM + Pyramid NRF) %
XS = calc_XCOM_XS(XCOM_atten,235.042);
XS = XS + calc_NRF_XS(centroid,FWHM,integrated_XS,'pyramid',energies);
masterCell{1,col_index} = 'U-235 XS (xcom + Pyramid NRF)';
masterCell(2:end,col_index) = num2cell(XS);
col_index = col_index + 1;

% Once all the cross-section columns are specified, it's time to write the
% cell array to a file for later use
writeDataToFile('masterArray.csv',',',masterCell);
```

calc_XCOM_XS.m

Calculates a scattering cross-section in barns from a mass attenuation coefficient and nuclear mass

```
% Takes the mass attenuation coefficient (in cm^2/g) from the XCOM database
% and converts it to a cross-section in barns
function XCOM_XS = calc_XCOM_XS(XCOM_mass_atten,A)
% XCOM_mass_atten is the mass attenuation coefficient in cm^2/g
% Note that XCOM generally does not have attenuation coefficients for
% the precise energy in question. Linear interpolation will get
% sufficiently close. Also note that these cross-sections do not change
% appreciably over small energy ranges.

% A is the nuclear mass number
Na = 6.022*10^23; % Avogadro's number
% Need to multiply by 10^24 to get XS in barns rather than cm^2
XCOM_XS = 10^24 * XCOM_mass_atten * A / Na;
```

Calc_NRF_XS.m

Computes NRF cross-sections using one of three supported models

```
% Computes an NRF cross-section as a function of energy, based on one of
% three models. Requires data on the centroid, FWHM, and integrated
% cross-section. energies is a array of energies to evaluate the
% function at.
function NRF_XS = calc_NRF_XS(centroid,FWHM,integrated_XS,mode,energies,N_pts)
% centroid and FWHM need to be in eV
% integrated_XS is in eV-barns
% mode can be one of three options, and selects the model of cross-section
% to use: 'square', 'gaussian', or 'pyramid'
% energies is the array of points to calculate the cross-sections for
% N_pts is the number of points to approximate the Gaussian shape with;
% used only in conjunction with mode = pyramid

% If N_pts is not given, we'll use the MCNP default of a 5-point pyramid
if nargin < 6
   N_pts = 5;
end

% Also, N_pts cannot be 1 (yields infinite slopes for the line segments).
% If N_pts = 2, we'll have a square function instead of a pyramid.
% We'll error if this happens.
if N_pts < 3
   error(['The number of points defining a "pyramid" must be greater than 2.']);
end
% Next, N_pts cannot be longer than the energies array, minus the two endpoints.
if N_pts > length(energies) - 2
   error(['The number of points defining the "pyramid" must be less than the number of points in "energies".']);
end
```

```
% Create the NRF XS array to be the same length as the energies array
NRF_XS = zeros(1,length(energies));

if strcmp(mode,'gaussian') == 1
   % The NRF cross-section is peaked at the centroid and is Gaussian in
   % shape

   % First, convert the FWHM to a standard deviation
   sigma = FWHM/(2*(2*log(2))^0.5);
   % Create the Gaussian function itself
   normalization = integrated_XS*(sigma*(2*pi)^0.5)^-1;
   NRF_func = @(E) normalization*exp(-(E-centroid)^2 /(2*sigma^2));

   % Now to evaluate this for every energy specified, and return the
   % result
   for i = [1:length(energies)]
      NRF_XS(i) = NRF_func(energies(i));
   end

elseif strcmp(mode,'square') == 1
   % The NRF cross-section is constant and nonzero near the centroid, and
   % zero elsewhere, such that it integrates to the appropriate total

   height = integrated_XS / FWHM; % The value of the constant XS

   % Now to set the XS for every energy specified, and return the result
   for i = [1:length(energies)]
      % If the energy is within FWHM/2 of the centroid, the XS is nonzero
      if i < length(energies) && energies(i+1) >= (centroid - FWHM/2) && energies(i+1) <= (centroid + FWHM/2)
         NRF_XS(i) = height;
      elseif i > 1 && energies(i-1) >= (centroid - FWHM/2) && energies(i-1) <= (centroid + FWHM/2)
         NRF_XS(i) = height;
      else
         NRF_XS(i) = 0;
      end
   end

   % Now to correct the height if the integration didn't come out quite right
   NRF_XS = NRF_XS * integrated_XS / (0.1*sum(NRF_XS));

elseif strcmp(mode,'pyramid') == 1
   % The NRF cross-section is peaked at the centroid, and consists of many
   % line segments. (An extension of the "pyramid" shape

   % First, let's compute the Gaussian
   % Convert the FWHM to a standard deviation
   sigma = FWHM/(2*(2*log(2))^0.5);
   % Create the Gaussian function itself
   normalization = (sigma*(2*pi)^0.5)^-1;
   NRF_func = @(E) normalization*exp(-(E-centroid)^2 /(2*sigma^2));
```

```
% Now to evaluate this for the number of points necessary, and construct the
% pyramidal shape. We need to figure out how to space the energies to
% make the desired number of points. linspace(x1,x2,N) does this.

% Defining the number of energies
E = [energies(1),linspace(centroid-4,centroid+4,N_pts),energies(end)];

% Corresponding values of the Gaussian
F = zeros(1,length(E));
for i = [1:length(E)]
   if i == 1 || i == length(E)
      F(i) = 0;
   else
      F(i) = NRF_func(E(i));
   end
end

% Let's make a cell of function handles representing the lines between
% the above points
lines = cell(1,length(E)-1);
for i = [1:length(lines)]
   if i == 1 || i == length(lines)
      m = 0; b = 0; % Flat line for ends of the pyramid shape
   else
      m = (F(i+1) - F(i))/(E(i+1) - E(i)); % Slope of the line
      b = F(i) - m*E(i); % y-intercept of the line
   end
   line{i} = @(x) m*x + b;
end

% Now to set the XS for every energy specified, and return the result
% We'll use the interpolateLine method to calculate XS values within
% the pyramidal section
% We need to fill out the pyramid shape by linearly interpolating
% between points.
NRF_XS = zeros(1,length(energies));

% The next loop goes through all the energies in the "energies" array,
% determines where they fall in the "E" array, and then picks the
% appropriate line for them.
area = 0; % The area under the approximate form of the Gaussian
for i = [1:length(energies)]
   % Now to determine the greatest lower bound of temp_E in the "E"
   % array.
   j = 1;
   while j < length(E)-1 && energies(i) >= E(j+1)
      j = j + 1;
   end
   % Without the clause below, the last two points use a flat line fit.
   % Only the first and last points should have zero cross-sections.
```

```
    if i == length(energies)-1
      j = j - 1;
    end
    funct = line{j};
    NRF_XS(i) = funct(energies(i));
  end

  % Finally, we need to normalize this pyramid to integrate to the
  % correct value. Let's sum all the areas under the curves.
  for i = [1:length(energies)]
    % Each section has rectangular base with a triangular top
    % (Except for the first line segment, which is unformly zero, from
    % the first to the second energies)
    if i > 1 && energies(i) < centroid % We're going to multiply the sum from this loop by 2 to get total area
      rectangle_A = (energies(i+1)-energies(i))*NRF_XS(i);
      triangle_A = (energies(i+1)-energies(i))*(NRF_XS(i+1)-NRF_XS(i))/2;
      area = area + rectangle_A + triangle_A;
    end
  end
  area = area*2; % Assumes the NRF cross-section shape is symmetrical about the centroid, which it is
  NRF_XS = NRF_XS * integrated_XS / area;

else % An incorrect mode was given
  error(['Choose a valid NRF cross-section model: square, gaussian, or pyramid.']);
end
```

---

interpolateLine.m

Estimates the value of a function using linear interpolation

```
function y3 = interpolateLine(x1,y1,x2,y2,x3)
% Linearly interpolates a value between two data points

% This function takes two points, (x1,y1) and (x2,y2), to construct a
% linear interpolation. This estimated function then returns a value (y3)
% for a third x-coordinate (x3)

m = (y2 - y1)/(x2 - x1); % Slope of the line
b = y1 - m*x1; % y-intercept of the line

line_eq = @(x) m*x + b;

y3 = line_eq(x3);
```

---

format_RTAB.py

Creates a compact version of an RTAB table

```
# This script removes all superfluous comments from an RTAB database
input_table = open("092_cs0sl_sm+nt.txt","r")
output_table = open("compact_92_sm+nt.txt","w")

def isPosInt(string):
	status = 1
	if isSciNotation(string) == 1:
		# Check scientific notation to be an integer
		number = float(string)
		if number - int(number) != 0:
			status = 0
	# isdigit() method tests if a string is only numeric. Only integers return true
	elif isSciNotation(string) == 0 and string.isdigit() == 0:
		status = 0
	elif int(float(string)) < 1:
		status = 0
	return status

# Python knows how to use scientific notation, but we have to convert the string into a float to use it
# This method checks to see if a string can be interpreted by Python as scientific notation
def isSciNotation(string):
	status = 1
	string = string.lower()
	if string.find("e"):
		try: float(string)
		except ValueError:
			status = 0
	else:
		status = 0
	return status

# Now to start scripting
for line in input_table:
   if line.startswith("*BLOCK:") == 1:
      output_table.write(line)
   elif line.startswith("   THETA") == 1:
      output_table.write(line)
   elif line == " \n":
      output_table.write(line)
   else:
      data = line.split()
      if data != [] and isPosInt(data[0]) == 1:
         output_table.write(line)
```

interpolate_RTAB.py

Linearly interpolates between two data sets in RTAB to estimates values at arbitrary energies

```
# This script takes data from a compact rtab table and estimates cross-sections for an arbitrary photon energy

############################
### FUNCTION DEFINITIONS ###
############################

def findNearestE(photon_energy,database):
	input_table = open(database,"r")
	# Initializing high/low energy values
	# high_diff and low_diff show how close in keV the energy is to the given photon energy
	high_E = -1; high_diff = -1
	low_E = -1; low_diff = -1
	for line in input_table:
		# If the line begins with *BLOCK: then we have found the start of a data section
		if line.startswith("*BLOCK:"):
			temp = line.split(":")
			temp = temp[1].split("keV")
			# Get rid of the non-numerical tail
			energy = float(temp[0])
			diff = abs(energy - photon_energy)
			# If either value is not yet defined, we'll define them.
			if high_diff < 0:
				high_diff = diff
				high_E = energy
			elif low_diff < 0:
				low_diff = diff
				low_E = energy
			# If we find closer energies, we must redefine high_E and low_E
			elif diff < high_diff or diff < low_diff:
				low_diff = high_diff; high_diff = diff
				low_E = high_E; high_E = energy
	if low_E > high_E:
		line = low_E
		low_E = high_E
		high_E = line
	input_table.close()
	return low_E,high_E

def findBoundingE(photon_energy,database):
	input_table = open(database,"r")
	# Initializing high/low energy values
	high_E = -1; low_E = -1
	for line in input_table:
		# If the line begins with *BLOCK: then we have found the start of a data section
		if line.startswith("*BLOCK:"):
			temp = line.split(":")
			temp = temp[1].split("keV")
			# Get rid of the non-numerical tail
			energy = float(temp[0])
			if energy < photon_energy and energy > low_E:
				low_E = energy
			if energy > photon_energy and high_E < 0:
				high_E = energy
```

```
	input_table.close()
	return low_E,high_E

def getDataForInterpolation(low_E,high_E,database):
	low_E_data = []; high_E_data = []
	input_table = open(database,"r")
	lowDataFlag = 0
	highDataFlag = 0
	for line in input_table:
		# Stop reading upon reaching the blank line separator between sections
		if line == " \n":
			lowDataFlag = 0
		elif lowDataFlag == 2: lowDataFlag = 1
		# Read cross-section data for low_E
		elif lowDataFlag == 1:
			low_E_data.append(line)
		# Start reading two lines after the *BLOCK: header line
		elif line.startswith("*BLOCK:" + str(low_E)) == 1:
			lowDataFlag = 2

		# Stop reading upon reaching the blank line separator between sections
		if line == " \n":
			highDataFlag = 0
		elif highDataFlag == 2: highDataFlag = 1
		# Read cross-section data for low_E
		elif highDataFlag == 1:
			high_E_data.append(line)
		# Start reading two lines after the *BLOCK: header line
		elif line.startswith("*BLOCK:" + str(high_E)) == 1:
			highDataFlag = 2
	input_table.close()
	return low_E_data,high_E_data

def interpolate(energy,low_E,low_E_data,high_E,high_E_data):
	interpolated = []
	# We're assuming that low_E_data and high_E_data are the same length
	# They should be, because RTAB gives data for each energy in a well-defined
	# angular distribution
	for index in range(0,len(low_E_data)):
		# Read in the data for each line
		# We will split it into numerical values next
		low_line = low_E_data[index]
		high_line = high_E_data[index]
		low_line = low_line.split()
		high_line = high_line.split()
		interpolated_line = []
		# First, add the angle value for this particular data
		interpolated_line.append(float(low_line[0]))
		# Now we'll find the rest of the data
		for j in range(1,len(low_line)):
			low_data_point = low_line[j].replace("\n","")
			high_data_point = high_line[j].replace("\n","")
			low_data_point = float(low_data_point)
			high_data_point = float(high_data_point)
			# Now for the linear interpolation!
			slope = (high_data_point - low_data_point)/(high_E - low_E)
			y_intercept = (low_data_point*high_E - high_data_point*low_E)/(high_E - low_E)
			interpolated_point = slope*energy + y_intercept
```

```
                        interpolated_line.append(interpolated_point)
                interpolated.append(interpolated_line)
        return interpolated

# Takes an arbitrary numerical value and returns it in scientific notation with a specified number of sigFigs
def formatValue(number,sigFigs):
        number = float(number)
        negativeFlag = 0
        if number < 0:
                number = abs(number)
                negativeFlag = 1
        # This gets our input into a standard format
        # Now we avoid errors with decimals like .234 vs. 0.234
        # float() also puts exponents in the right form for number < 0.0001
        string = str(number).upper()
        if string.find("E") == -1: # Not yet in scientific notation
                index = 0
                exp = 0
                if string.find(".") >= 1 and string.startswith("0") == 0: index = string.find(".")
                elif string.find(".") <= 1: # A decimal < 1
                        # We must find the first nonzero digit
                        # Since we already converted to a float in the beginning, we are limited
                        # to decimals > 0.0000999999...
                        if len(string) > 3 and string[2] != "0": index = -1
                        elif len(string) > 4 and string[3] != "0": index = -2
                        elif len(string) > 5 and string[4] != "0": index = -3
                        elif len(string) > 6 and string[5] != "0": index = -4
                else: index = len(string) # For numbers without a decimal point (ints)

                if index > 0: exp = index - 1
                else: exp = index
                coeff = number/10**exp

                # If exp is a single digit, we will add a "0" in front, i.e. 1.0e-2 -> 1.0e-02
                if abs(exp) < 10:
                        # Must take into account a negative sign
                        if exp < 0:
                                exp = "-0" + str(abs(exp))
                        else: exp = "0" + str(exp)
                if int(exp) >= 0 and str(exp).find("+") == -1:
                        exp = "+" + str(exp)

                # We also want to round the coefficient to an appropriate number of sig figs
                coeff = str(coeff)
                if coeff.find(".") == -1: coeff = coeff + "."
                if len(coeff) > sigFigs + 1: # +1 to account for decimal point in the string
                        coeff = coeff[0:sigFigs+1]
                else:
                        while len(coeff) <= sigFigs:
                                coeff = coeff + "0"
                string = coeff + "E" + exp
        else: # we need to now make sure of the number of sig figs
                temp = string.split("E")
                coeff = temp[0]
                if coeff.find(".") == -1: coeff = coeff + "."
                if len(coeff) > sigFigs + 1: # +1 to account for decimal point in the string
                        coeff = coeff[0:sigFigs+1]
                else:
```

```
                        while len(coeff) <= sigFigs:
                                coeff = coeff + "0"
                string = coeff + "E" + temp[1]

        if string.find("E-02") > -1 and number > 0.1:
        print(number)
        print(string)
        print("_____")
        if negativeFlag == 1:
                string = "-" + string
        return string


##################
### SCRIPTING ###
##################

photon_energy = 1408.1
database = "92_sm+nt_new.txt"

# First, let's find which RTAB tables are closest to the energy in question
# This should provide the best interpolation
(low_E,high_E) = findBoundingE(photon_energy,database)

# If we couldn't find data with low_E < photon_energy < high_E, then we'll just use the two nearest values
if low_E < 0 or high_E < 0:
        (low_E,high_E) = findNearestE(photon_energy,database)

print("Energies used for interpolation/extrapolation")
print("LOW: " + str(low_E) + " keV")
print("HIGH: " + str(high_E) + " keV")

# Now we want to load in the data from the two nearest-energy tables
(low_E_data,high_E_data) = getDataForInterpolation(low_E,high_E,database)

# We have the data, so it's time for the interpolation
newData = interpolate(photon_energy,low_E,low_E_data,high_E,high_E_data)

# Finally, let's output the interpolated data into its own table file
# We can reintegrate this in a new RTAB file later
output_table = open("interpolated_data.txt","w")

# Let's start with the header rows
if high_E < photon_energy: output_table.write("*BLOCK:" + str(photon_energy) + "keV    (Extrapolated from " + str(low_E) +
"keV and " + str(high_E) + "keV data)\n")
else: output_table.write("*BLOCK:" + str(float(photon_energy)) + "keV    (Interpolated from " + str(low_E) + "keV and " +
str(high_E) + "keV data)\n")
output_table.write("   THETA    CS(B/SR)     X(1/A)  A-PARALLEL-R0 (RE,IM)   A-PERPENDICULAR-R0 (RE,IM)\n")

for entry in newData:
        currentLine = ""
        for index in range(0,len(entry)):
                # We want to convert everything to scientific notation with six signficant digits
                if index > 0: value = formatValue(entry[index],6)
                else:
                        value = str(entry[index]) # The first entry is always an angle
                        # Want to make sure the angles have 3 zero decimal points
                        temp = value.split(".")
```

```
                        value = temp[0] + ".000"
                        # Next, we ensure the spacing is even
                        while len(value) < 7:
                                value = " " + value

                spacingChar = " " # Ensures that all values take the same number of spaces
                if value.startswith("-") == 0 and index > 0: spacingChar = "  "
                currentLine = currentLine + spacingChar + value
        output_table.write(currentLine + "\n")
```

---

insert_RTAB.py

Writes interpolated RTAB data into the larger RTAB data file

```
# This script inserts new RTAB data into a compact rtab table

############################
### FUNCTION DEFINITIONS ###
############################

def findE(filename):
    new_table = open(filename,"r")
    for line in new_table:
        # If the line begins with *BLOCK, we can read off the energy
        if line.startswith("*BLOCK:"):
            temp = line.split(":")
            # temp[1] is the rest of the line after *BLOCK
            temp = temp[1].split("keV")
            # Now temp[0] is the energy
            energy = temp[0]
            break
    new_table.close()
    energy = float(energy)
    return energy

def findBoundingE(photon_energy,database):
        old_table = open(database,"r")
        # Initializing high/low energy values
        high_E = -1; low_E = -1
        for line in old_table:
                # If the line begins with *BLOCK: then we have found the start of a data section
                if line.startswith("*BLOCK:"):
                        temp = line.split(":")
                        temp = temp[1].split("keV")
                        # Get rid of the non-numerical tail
                        energy = temp[0]
                        energy = float(energy)
                        if energy < photon_energy and energy > low_E:
                                low_E = energy
                        if energy > photon_energy and high_E < 0:
                                high_E = energy

        old_table.close()
        return low_E,high_E

def insertData(energy,low_E,insertData,oldData,newData):
    insert_table = open(insertData,"r")
    old_table = open(oldData,"r")
    new_table = open(newData,"w")
```

```
    low_E_flag = 0 # Sets to 1 if we've read to the low_E entry in oldData
    for line in old_table:
        new_table.write(line)
        if line.startswith("*BLOCK:" + str(low_E)) == 1:
            low_E_flag = 1

        # The first blank line tells us to write the insert data
        if low_E_flag == 1 and line == " \n":
            low_E_flag = 0
            for entry in insert_table:
                new_table.write(entry)
            new_table.write("\n")
    insert_table.close()
    old_table.close()
    new_table.close()

##################
### SCRIPTING ###
##################

# For what energy is the new data?
energy = findE("interpolated_data.txt")

# Between which entries will we place the new data?
(low_E,high_E) = findBoundingE(energy,"compact_92_sm+nt.txt")

# Now that we know the limits, let's insert the data
insertData(energy,low_E,"interpolated_data.txt","compact_92_sm+nt.txt","new_92_sm+nt.txt")
```